Questions Raised During Webinar on 18th October 2011

Chris Hobbs
(with input from Justin Moon)
QNX Software Systems
chobbs@qnx.com

October 2011

1 Introduction

On 18th October 2011, QNX and EE Times held a joint webinar on the subject "When COTS is not SOUP: Commercial Off-the-Shelf Software". During the webinar a number of questions was raised and, although some were answered verbally or textually during the webinar, the available time prevented them all being addressed. This document lists all of the questions asked during the webinar and provides answers for them.

To continue the discussion, please free to contact QNX at jmoon@qnx.com.

2 General IEC 62304 Questions

Question 1 *SOUP = Software of Unknown Provenance?*

That's the definition given in IEC 62304. I have also heard, and sometimes accidently use, "Software of Uncertain Pedigree", " Software of Unknown Pedigree" and "Software of Uncertain Provenance".

As I said during the presentation, IEC 62304 defines it as either software that was not designed for use in the/a medical device or software for which the development process is unknown.

Question 2 *Inhowfar is IEC 62304 compatible with agile (rather than waterfall) processes?*

IEC 62304 does not mandate the use of any particular development methodology. Section 5.1.1, note 2, specifically states:

> . . . activities and tasks may overlap or interact and may be performed iteratively or recursively. It is not the intent to imply that a waterfall model should be used.

There is a significant history of successful, and safe, medical products developed by agile techniques. The key word is "discipline": the actual methodology used is less important than the discipline within the development team applying the process. "Agile" is not a euphemism for a lack of documentation or traceability. Define the process you will be using, demonstrate through a hazard and risk analysis that it is suitable and then apply it in a disciplined manner.

Question 3 *Given the example on slide 8 and your emphasis that testing is not the answer, what is?*

That is a good question. And a difficult question. I believe that, as the efficacy of testing is dropping for the reasons I gave during the webinar, the efficacy of (deep) static analysis and design validation is increasing.

A design validation tool like SPIN finds the problem with the simple, two-thread program on slide 8 in under a second on my computer—it finds a sequence of 90 steps that can lead to x having a value of 2 at the end of the program. I think that these types of tool, based on Temporal Logic, should be used to prove the correctness of the design of any protocol or algorithm. Using this type of tool retrospectively, basing its model on the actual code rather than the design, can also pick up implementation errors. There are numerous papers around that illustrate how this type of validation can uncover very subtle bugs: see, for example, "Formal Analysis of a Space Craft Controller using SPIN" by Klaus Havelund, Mike Lowry and John Penix.

As far as static checking of code is concerned, LINT-like, shallow tools have given this technique a bad name because of the large number of false positives they produce. However, there are more sophisticated tools that work with semantic as well as syntactic knowledge of the programming language (e.g., Coccinelle) and symbolic execution tools that work on the boundary between static and dynamic analysis (e.g., klee).

I have mentioned only open source tools in this answer, not because there are not good COTS tools but because I wouldn't want to favour a few of these.

Question 4 *What about the standard C library that comes with the compiler? Is that considered SOUP? Seems like it is impossible to get away from using it, but you may not have access to all the records behind it.*

Having recognised this as a potential problem, QNX included the whole of `libc` in its IEC 61508 SIL3 certification. We make no restrictions on the use of any of the `libc` functions.

Question 5 *There is a problem with software SILs in that some certifying bodies interpret "SIL" to be a measure of the rigour of the software life-cycle rather than a measure of its probability of failure. So how can I be sure of the certificate that accompanies the COTS?*

IEC 61508 requires (or "recommends" or "strongly recommends") a number of different things:

1. Various aspects of the process to be used. For example *"The design method chosen shall possess features that facilitate software modification. Such features include modularity, information hiding and encapsulation"*.

2. Various techniques to be used (or avoided): *"Online checking of the installation of dynamic variables"*.

3. The numerical values of failure probability for the different Safety Integrity Levels (SILs). For example, for a SIL3 device that is required to function continuously, the probability of failure per hour of operation is to be less than 10^{-7}.

4. Acceptable statistical techniques for justifying failure rates. For example, given n independent test cases applied to an on-demand system, the failure probability per demand can be assumed to be $p \leq 1 - \sqrt[n]{\alpha}$ at confidence level α.

Given that the whole of part 5 of IEC 61508 (*Examples of methods for the determination of safety integrity levels*) is dedicated to techniques that can be used to justify a particular safety claim, it would seem perverse for an auditor to ignore items 3 and 4 and concentrate the audit on item 1. The fact that this question was

raised (by an experienced auditor) is perhaps a warning to companies intending to incorporate SOUP that has been certified to a Functional Safety standard into their medical product, to ensure that the certification that comes with the SOUP includes validation by the auditor of the claimed Safety Integrity Level.

Question 6 *What operating systems do you recommend?*

The short answer is, of course, that I recommend the QNX Neutrino RTOS. The more general answer is that I would look for an operating system with the following characteristics:

- It minimises the amount of code that runs in supervisor mode. Code running in supervisor mode is intrinsically dangerous and, although some code has to run in this mode, this should be minimised. In particular, communication stacks and file-systems do not need to run in supervisor mode.

  This will help me find an architecture that meets my dependability requirement because, if these components crash, they will not affect the stability of the operating system and it will be possible to restart them cleanly.

- It allows drivers to run in non-supervisor mode. This is really an extension of the point above but drivers, in particular, can cause instability in systems.

  User-mode drivers allow me greater flexibility for debugging and their use again also reduces the probability of system instability: if they fail then they can be restarted.

- It allows code of different safety levels to be run—see question 14.

  This is important because not only is the cost of writing and verifying all the software in the system to the highest standards unnecessarily costly, it is also discouraged by the standards.

- It has a numerical claim of dependability that has been verified by an external and independent auditor.

  This allows me to incorporate it into my failure model easily.

Question 7 *Is there a published list of information for common COTS (e.g. MS Windows) that helps in documenting clear SOUP?*

I'm afraid that I don't know of one.

Question 8 *So how do you recommend approaching things that are minor but unclear SOUP? For instance, we have a touch-screen for which the vendor provides. They are nearly always consumer electronics driven and have little interest in providing extra information for low quantity medical electronics.*

A lot depends on:

- what a failure of the component would have on the overall dependability of the system. This would presumably be measurable from the fault tree for your product.
- whether the vendor provides you with source code or only an executable.

If the failure of the component would not itself affect the continuing safe operation of the product (e.g., a logging system) then presumably the concern is whether its failure could indirectly affect a more critical component (e.g., failure of the logging system causing a safety-critical component to fail when it tries to write a log message). Such dependencies would have to be traced and corrected. See also the answer to question 14.

If the component provides part of the safety-critical operation of the system, then more work would be required. If you have access to the source code then you could incorporate the component into your own process and carry out your normal code inspections, static analysis, etc. In this case you are effectively taking ownership of the code and treating it as though you had yourself written it.

If you don't have access to the source code or the source code is effectively impenetrable, then perhaps you should be questioning whether its inclusion in a medical device is appropriate. However, some level of confidence can be gained by random testing. IEC 62304 doesn't provide guidance on the number of tests required but IEC 61508 does and an auditor would probably be willing to accept guidance from a more stringent specification, particularly as IEC 61508 is is mentioned in IEC 62304 as "inspiring" the implementation of medical device software (figure C.1). Appendix D *(A probabilistic approach to determining software safety integrity for pre-developed software)* of Part 7 of IEC 61508 (2010) provides guidance on the number of such tests that would need to be performed for different confidence levels. You could then say to an auditor,

> *"Using the mechanism recommended in IEC 61508, we have demonstrated to a confidence level of 98% that the component will react correctly at least 9,999 times out of 10,000. We have incorporated*

that figure into our failure model for the entire device and, as you can see, the final result for the whole system is acceptable".

One other point to consider if the vendor is not creating the component specifically for safety-critical applications is how you will handle bug fixes and upgrades published by the vendor. Will you be able to accommodate these into your certified product in a controlled manner?

Question 9 *What kind of certification should be provided for an OS intended to be used for a Medical Device?*

At the moment, I frankly don't know. We are working with various certification organisations to determine this.

Question 10 *Are there International Agencies recognized everywhere in the world for software certification?*

There are many organisations world-wide that can provide software certification. The process is that a national body (e.g., UKAS in the United Kingdom, Deutsche Akkreditierungsstelle GmbH (DAkkS) in Germany) *accredits* a company to *certify* a particular product or process to a particular level. Thus a certification company will examine your software and issue a certificate to indicate that you are compliant with a particular standard. That certification company should have an accreditation from its national body effectively saying that it's competent to issue the certificate.

Actually, there is nothing to stop any company setting up to do certification without national accreditation but the certificate that that company issues will generally not be worth as much (perhaps not as acceptable to your customers).

So, a company might be accredited, for example, to certify a software product to Safety Integrity Level 2 (SIL2) in accordance with IEC 61508.

To date, I don't know of any organisation that is accredited to certify a product to IEC 62304 but that doesn't mean that there aren't any.

Question 11 *How is the selected hardware (micro-controller, etc) affected by 62304? How can one start with the selection process for HW/OS/SW for a "Class C" product?*

IEC 62304 is entitled "Medical Device Software — Software Life-cycle Processes". It thereby specifically addresses software, excluding hardware. Other

standards such as IEC 60601 and IEC 61010 (for laboratory equipment) cover the development of the entire medical device and would need to be applied to determine the suitability of a particular processor.

The more general part of this question, how one starts with the selection process, is also interesting. When I am consulting with customers, I tend to stress the "three Es" from *Software for Dependable Systems: Sufficient Evidence?*:

1. **Explicit claims.** No system can be "dependable" in all respects and under all conditions. So to be useful, a claim of dependability must be explicit. It must articulate precisely the properties the system is expected to exhibit and the assumptions about the systems environment upon which the claim is contingent. The claim should also indicate explicitly the level of dependability claimed, preferably in quantitative terms. Different properties may be assured to different levels of dependability.

2. **Evidence.** For a system to be regarded as dependable, concrete evidence must be present that substantiates the dependability claim. This evidence will take the form of a dependability case arguing that the required properties follow from the combination of the properties of the system itself (that is, the implementation) and the environmental assumptions. Because testing alone is usually insufficient to establish properties, the case will typically combine evidence from testing with evidence from analysis. In addition, the case will inevitably involve appeals to the process by which the software was developedfor example, to argue that the software deployed in the field is the same software that was subjected to analysis or testing.

3. **Expertise.** Expertise—in software development, in the domain under consideration, and in the broader systems context, among other things—is necessary to achieve dependable systems. Flexibility is an important advantage of the proposed approach; in particular the developer is not required to follow any particular process or use any particular method or technology. This flexibility allows experts freedom to employ new techniques and to tailor the approach to the systems application and domain. But the requirement to produce evidence is highly demanding and likely to stretch todays best practices to their limit. It will therefore be essential that developers are familiar with best practices and deviate from them only for good reason.

That tends to structure the "how do I get started?" discussion: first make the claims and the environmental assumptions explicit in writing. This is not easy and changes may have to be made to the claims later but they provide the framework within which the architecture, design and implementation must proceed. Setting the claims is harder with a standard such as IEC 62304 than it is with IEC 61508

because you have to create the numerical values on dependability yourself, based on the clinical use of the device (and the class that is being claimed).

Once the claims are explicit, they can be accommodated into a system-level failure model and this can be used to provide various budgets[1] for the components to be used: e.g., the system-level dependability budget requires a dependability of x for component Y. This provides the constraints within which component selection can be carried out. Processor Y cannot provide the required level of dependability and so either the architecture has to be changed (e.g., to provide a Safety Bag) or a different processor has to be chosen.

The other result of performing the budgetary analysis is that the second of the three Es is being prepared *en passant*: the evidence that will be needed to support the claims when the entire Safety Case is prepared.

I have seen customers who have produced a hardware design without consideration of the software and, by virtue of the hardware restrictions, have made it impossible to create a device with the necessary dependability. This is why it is important that these first steps be made for the whole system.

The third E is perhaps the most important and the least easy to solve. There are very few software architects with the necessary tools to perform these sorts of analyses.

Question 12 *You haven't said how the use of SOUP changes depending on whether the system is class A, B or C. Is a process like this important for class A devices?*

The level of dependability that would be needed from the SOUP component would be determined by the budget allocated from the system-level failure model (see question 11). While the system dependability predicted by that model would presumably depend on whether the device was to operate at class A, B or C, once the required dependability of the SOUP component had been determined, it would not matter for the component what the device class was.

This emphasises the importance of determining the claims and building the failure model right at the beginning of the development process.

Question 13 *Does "MISRA" exist for medical devices?*

MISRA defines coding standards for software for use in automobiles (effectively creating a subset of the language that avoids the worst pitfalls—e.g., undefined

[1] These may include performance budgets, dependability budgets and even cost budgets but this answer concentrates on the dependability budget.

commands in C (a[i] = i++)). I don't know of an equivalent for medical devices although I would expect that an auditor would look kindly on any defined subset, such as the MISRA one, being used.

Question 14 *What about segmentation of a system into "safe" and "more commercial" pieces? For instance: data acquisition in RTOS or assembly and an OS for "data handling" (much like loading data from an instrument and looking at in in MS Excel, etc...).*

This is something we're meeting all the time. Most systems contain software modules from various sources, developed under different processes and with different demands on their dependability. For example, a medical system might contain both a logging application and a module that calculates the drug flow required to complete a patient's treatment. Although the logging application is important, its failure would not create a hazard for anyone. However, a failure of the drug flow control module could compromise the patient's treatment, or even kill her.

In many cases, separating the functions out onto different hardware would actually make the problem more acute by introducing additional interfaces.

It is essential that such non-critical and critical modules be isolated from each other so that no behaviour of non-critical modules could possibly affect critical modules. Similarly critical modules must be isolated from each other. This isolation has three parts:

1. Physical Isolation. This requires that the operating system provide strong memory isolation. For example, it is important that when a message is copied from a sender to a receiver, there is no window when it is write-accessible by both.

2. Resource Isolation. This means that the modules must not be able to consume resources (file descriptors, memory, mutexes, etc.) in such quantities that the critical modules are starved and unable to perform their function.

3. Temporal Isolation. One resource that must be available to the critical modules is processor time: it must not be possible for non-critical or other critical modules to absorb so much processor time as to starve a particular critical module.

The QNX Neutrino RTOS Safe Kernel supports these three types of isolation. In particular, it provides temporal isolation through the use of the Adaptive Partitioning Scheduler (APS).

Note that not only can it be very time-consuming and expensive to develop code that is non-critical to the same standard as critical code, but that this practice is discouraged by many standards. For example, IEC 61508 and EN 50128 specifically require that the amount of critical code be reduced to a minimum and good isolation be provided between critical and non-critical code. Section 9.4.8 of EN 50128 states:

> The software architecture shall minimise the safety part of the application.

3 QNX-Specific Questions

Question 15 *Can you provide the URL to the testing whitepapers on qnx.com?*

Go to `http://www.qnx.com/` and click on Downloads→All. A link to whitepapers will then appear on then left of the screen and, once this is selected, you can access whitepapers on different topics.

Question 16 *Is all of the QNX product 62304 compliant? i.e. Photon, PPS, mqueue etc.*

The QNX medical offering provides the core operating system that is most critical to the customer's applications. Peripheral OS services such as Photon and network stacks are not part of QNX's certified solutions. Mqueue is an experimental feature that is not officially supported so I would discourage its use altogether. PPS is also beyond the scope of the certified solutions.

Question 17 *Can QNX use previously FDA (510K or PMA) approved COTS Software as a predicate for subsequent submissions? If so, can Medical OEM's use this as an advantage?*

The current stance is the OS is simply another component vendor. If a vendor uses the OS on a product and then creates a similar product with the same stack but with additional features they can file for 510k premarket notification which assures the device is similar to something already on the market, leverage all of the previous testing, documentation etc. in the new product. SOUP tests for the new features, requirements tracking for them, etc. and everything before could be leveraged. From an approval process, things would have to be resubmitted.

Question 18 *How about customers who have thousands of units with QNX versions prior to 6.5 and are unable to upgrade their OS? What can QNX do to help them with adherence to 62304?*

The compliance statement work is derived from the work we completed with IEC 61508 SIL3 certification. However as part of our commercial initiatives we can provide a package that contains evidence such as proven-in-use data to support for other versions of Neutrino. This package is designed to help a customer with an approvals process.

As was explained during the webinar, using clear SOUP (a pre-certified component) is the most cost-effective way to system certification.

Question 19 *There are three versions of the QNX kernels. Are they truly different? Is there regular kernel less safe or less qualified than the secure kernel or safe kernel?*

If you are asking about the QNX Neutrino Kernel 6.5.0 and QNX Safe Kernel, then the answer is: they are technologically similar. However, just as your certified product would mean a huge investment on your part, our Safe Kernel bears a lot of weight and carries our customers a long way when it comes to certification. From that sense, the two products are not similar. Try to incorporate the Neutrino Kernel 6.5.0 in your overall system and build a safety case without an OS-level certificate, you will see how much more it will cost you.

Question 20 *Does QNX have an external audit performed for the intended use as part of a MEDDEV?*

An independent and external auditor has reviewed the processes under which the Safe Kernel was implemented and the risk analysis that was performed and has concluded that the Safe Kernel is a suitable operating system for medical devices being developed in accordance with IEC 62304.

4 House-Keeping Questions

Question 21 *Are these slides available for download?*

Slides are available in pdf format by launching the on-demand version of this webcast and clicking on the “Additional Resources” button.

5 The Most Important Question

Question 22 *Is it difficult to find an accompanist for Schubert Lieder?*

Very. This question stems, I suspect, from my CV (or "resumé" for North American readers) that must have accompanied the webinar details. I have been working for the last decade on Schubert's Winterreise song cycle with dips backwards to the Schöne Müllerin and forwards to Schwanengesang. An hour wrestling with those is like a hour in the gym. My wife accompanies me but admits that the piano' is not her first instrument. Any offers?